Real-Time Tasks in SDL

Dennis Christmann and Reinhard Gotzhein

Networked Systems Group
University of Kaiserslautern, Germany
{christma,gotzhein}@cs.uni-kl.de

Abstract. SDL is a formal design language for distributed systems that is also promoted for real-time systems. To improve its real-time expressiveness, several language extensions have been proposed. In this work, we present an extension of SDL to specify *real-time tasks*, a concept used in real-time systems to structure and schedule execution. We model a real-time task in SDL as a hierarchical order of executions of SDL transitions, which may span different SDL processes. Real-time tasks are selected for execution using time-triggered and priority-based scheduling. We formally define real-time tasks, show their syntactical and semantical incorporation in SDL, present the implementation approach in our SDL tool chain, and provide excerpts of a complex MAC protocol showing the use of real-time tasks in SDL.

1 Introduction

The Specification and Description Language (SDL) [1] is a formal design language for distributed systems. It has matured and been applied in industry for several decades. SDL is also promoted for real-time systems. By its notion of time (`now`) and its timer mechanism, SDL provides significant, yet limited real-time expressiveness. Some real-time extensions have been defined as part of a dialect called SDL-Real-Time (SDL-RT) [2], and there is also tool support for tight integration of code generated from SDL specifications with existing real-time operating systems [3,4]. In this paper, we revisit the design of real-time systems with SDL and propose an extension to specify real-time tasks.

A real-time system is a reactive system in which the correctness of the system behavior depends on the correct ordering of events and their occurrence in time (see, e.g., [5]). Execution of real-time systems is usually decomposed into execution units called *real-time tasks* (or *tasks*[1] for short), which are scheduled according to their urgency. Tasks may be initiated when a significant change of state occurs (event-triggered) or at determined points in time (time-triggered). For predictable timing, it is important to determine worst case execution times (WCETs) of tasks.

Following Kopetz [5], a task is a sequential code unit executed under the control of the local operating system. In SDL, a code unit could be associated with

[1] Not to be confused with tasks, i.e., (sequences of) statements, in SDL.

Ø. Haugen, R. Reed, and R. Gotzhein (Eds.): SAM 2012, LNCS 7744, pp. 53–71, 2013.

an SDL transition. Correspondingly, an execution unit could be defined as SDL transition that is executed by an SDL engine. However, this is not sufficient for the general concept of real-time tasks in SDL, because system functionalities are often not performed sequentially by a single execution unit but are distributed across several SDL transitions. Hence, we adopt a more general concept of task in this paper: A *real-time task* has one defined starting transition execution and may then fork one to many subsequent and/or concurrent transition executions recursively. Formally, this concept is captured by a hierarchical order of transition executions. SDL transition executions can be associated with one or more SDL processes; hierarchical execution ordering can be achieved by exchanging SDL signals. Note that this allows the same SDL transition to be executed as part of different tasks, a degree of freedom that we consider as crucial. Real-time tasks may be triggered by time or by events, and have a scheduling priority, which determines the local order of execution units if several tasks are active at the same time. Time-triggered execution could be specified with SDL timers.

In our previous work, we have identified ways to augment SDL's real-time capabilities. In particular, we have proposed the following extensions supporting restricted forms of real-time tasks:

- In [6], we have introduced the concept of *SDL real-time signal*, which is an SDL signal for which an arrival time is specified when the signal is sent. The signal is transferred to its destination as usual, and appended to its input queue. However, consumption of the signal is postponed until the specified arrival time. The concept of real-time signals has been adopted in SDL-2010 [1] by adding activation delays to signal outputs. Beside SDL timers, real-time signals state a second way to activate time-triggered tasks in SDL.
- In [7], we have proposed *SDL process priorities* combined with a mechanism to suspend and resume SDL processes, with the objective to achieve short or even predictable reaction delays. In our experiments, we have shown that reaction delays of SDL processes can be substantially shortened. However, this does not reflect the general structure of tasks, which may span several SDL processes and/or share common SDL transitions. Therefore, process-based scheduling of SDL systems is not sufficient for many real-time systems.

In general, real-time tasks enhance SDL specifications by providing information on the dynamics at the system's runtime. Thus, they go beyond existing scheduling approaches that are based on static components like SDL transitions or SDL processes [7,8]: First, by grouping – possibly process-spanning – functionalities, real-time tasks are a structural concept that is orthogonal to SDL systems, and are not limited to a 1:1-correspondence between task and SDL transition. Consider, for instance, an SDL process realizing a communication protocol entity. Obviously, transitions of this process can be executed to transfer messages of different applications, thereby belonging to different tasks. Second, tasks have a scheduling priority, which is not supported in SDL and can also not be emulated by existing scheduling extensions, because they all rely on static system structures without comprehension of dynamic, distributed, and transition-sharing functionalities.

In this paper, we incorporate the general concept of real-time task into SDL. More specifically, we formally define SDL real-time tasks, and outline required syntactical and semantical extensions in Sect. 2. In Sect. 3, we discuss how they can be implemented in our SDL tool chain, consisting of SDL compiler, SDL runtime environment, and environment interfacing routines. In Sect. 4, we show excerpts of a complex MAC protocol to demonstrate the use of SDL real-time tasks. Section 5 surveys related work. Finally, Sect. 6 presents our conclusions.

2 Real-Time Tasks in SDL

In this section, we introduce the concept of real-time tasks in SDL, thereby providing a „language tool" to group functionally related behavior. Particularly, we argue for a transition-spanning notion of real-time task and for the scheduling of these tasks according to their urgency. We formally define real-time tasks (Sect. 2.1), incorporate them in SDL (Sect. 2.2), and present the required extensions of the SDL syntax (Sect. 2.3). Corresponding modifications of the formal SDL semantics can be found in the Appendix.

2.1 Formalization of Real-Time Tasks in SDL

We formalize real-time tasks in SDL by associating a set of transition executions with each task, and by defining a hierarchical order between them. This means in particular that an SDL real-time task has a starting point, which is the first transition execution, and may then spawn further transition executions in an iterative way. Furthermore, transition executions that are not ordered may occur concurrently. A transition may be executed several times as part of the same task. The same transition can also be executed by several tasks (transition sharing). An SDL real-time task terminates as soon as all transition executions have terminated. The set of transition executions is determined at runtime, depending, e.g., on the states of SDL processes.

Definition 1. *A real-time task τ is a tuple (τ_{id}, $T_e(\tau)$, f_{prio}, $<_{eo}$), where τ_{id} is a unique task id, $T_e(\tau)$ is the set of transition executions, $f_{prio} : T_e(\tau) \to \mathbb{N}$ is a function assigning a priority to each transition execution, and $<_{eo} \subsetneq T_e(\tau) \times T_e(\tau)$ is an execution order, which is a hierarchical order on $T_e(\tau)$:*

- *$<_{eo}$ is irreflexive, transitive, and antisymmetrical*
- *$\exists t_e \in T_e(\tau).\forall t'_e \in T_e(\tau).(t'_e \neq t_e \Rightarrow t_e <_{eo} t'_e)$, i.e. there is a smallest element defining the starting point of the task, which is the first transition execution.*

Note that the definition of real-time tasks allows the execution of particular sub tasks with different priorities. In SDL, transitions can only be executed if all firing conditions (process state, input signal, enabling condition) are satisfied. This means that even if all transition executions preceding an execution t_e have occurred, t_e may still be delayed. Also, the signal triggering t_e may be discarded as result of an implicit transition in a different state. So, to achieve sufficiently

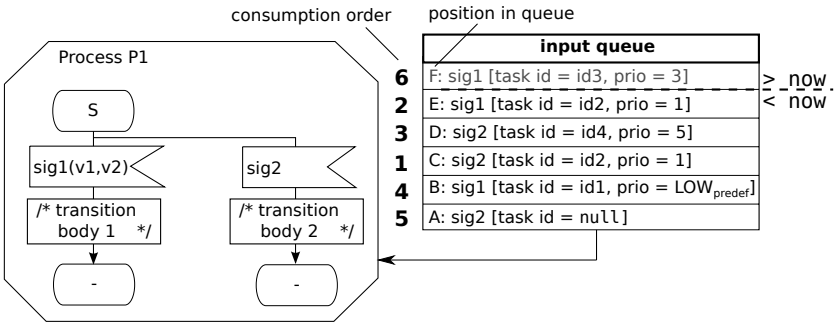

Fig. 1. Implications of real-time tasks to the selection of transitions

predictable execution times of real-time tasks, additional considerations at design time are required.

We furthermore classify real-time tasks regarding their activation paradigm.

Definition 2. *A real-time task is time-triggered, if the first transition is either triggered by a timer instance or by a signal with given activation delay. Otherwise, it is event-triggered.*

Note that an event-triggered SDL task may have time-triggered transition executions by using signals with activation delays or SDL timers.

2.2 Incorporation of Real-Time Tasks in SDL

To incorporate real-time tasks in SDL, we dynamically associate transition executions with task attributes consisting of task ids and task priorities. Thereby, the same SDL transition may be executed in several tasks, and be scheduled with different priorities. Furthermore, we introduce task signals, which extend plain SDL signals and SDL timer signals by task attributes. When consuming a signal, the signal's task attributes are assigned to the execution of the corresponding transition, thereby running the transition in the context of that task. Thus, task signals are used both to trigger task executions and to dynamically associate transitions and tasks at runtime.

Figure 1 shows an example with two transitions of an SDL process P1 and the current state of its input queue. The input queue holds five task signals and one plain SDL signal without associated SDL task. As in standard SDL, the signals have been inserted in FIFO order according to their availability time. This order is illustrated by using the characters from A (lowest availability time) to F (highest availability time). When determining the consumption order of the signals, their task attributes are additionally evaluated. In the absence of task signals, SDL signals are consumed as in standard SDL. Thereby, the extensions are compatible to the standard. If task signals are available, they have preference over plain SDL signals and are consumed according to their task priority. The resulting order of the example is given by numbers 1 to 6 in Fig. 1.

Because task signals are preferential, `sig2` at position A is not consumed first. Instead, `sig2` at position C with task id `id2` is taken to trigger the first transition execution, since it is the first signal in the input queue with highest task priority (lowest integer value). Afterwards, signal `sig1` at position E is consumed, which has the highest remaining task priority. According to the task priority, the next signal would be `sig1` at position F, which is, however, ignored, because the transition execution is time-triggered and the signal's availability time is larger than the current system time. Instead, `sig2` at position D, the first task signal with the next higher task priority, is consumed. The fourth signal is `sig1` at position B, which has the lowest possible task priority that is assigned if no task priority is defined explicitly (see also Sect. 2.3). Now, `sig2` at position A is consumed, because there is no available task signal left. Finally, `sig1` at position F is removed from queue as soon as it becomes available.

With real-time tasks, the language expressiveness of SDL is improved. In particular, we point out that it is not possible to map real-time tasks and task priorities to SDL-2000 [9], or to SDL-2010 [1], which introduces signal priorities and multi-level priority inputs. First, there is no equivalent notion of real-time task in standard SDL, i.e. sets of signals and transition executions can not be grouped and assigned to a specific functionality. Second, there are no mechanisms to let a transition creating signals define the signals' urgencies and to adequately influence their consumption order at the receiver. Signal priorities are not sufficient for this, because they do not take precedence over the signals' availability time. With priority inputs, on the other hand, the state of the receiver and not the urgency of the signal defines the consumption order. Hence, it is, for instance, not possible with standard SDL to achieve the same transition execution order as in Fig. 1, because standard SDL consumes signals of the same type always according to their position in the input queue.

2.3 Syntactical Extensions of SDL

Implications of real-time tasks to the consumption order of signals require several semantical extensions (see Appendix). To create real-time tasks, extensions of the SDL syntax become necessary as well. These modifications are given in List. 1.1 and are based on the syntax in Z.101 (Basic SDL) of SDL-2010 [10].

To control real-time tasks, we introduce task actions with new keywords **newTask** and **contTask** (line 4 in List. 1.1). Both can be optionally specified when signals are created, i.e. in output and set actions (lines 1 and 2). Specifying **newTask** denotes a *task creation* and using **contTask** causes a *task forking*, which continues an existing task. The created/continued task is associated with the generated signal, which has the role of a task trigger. The task is activated (**newTask**) or continued (**contTask**) as soon as the signal is consumed. If the signal has an activation delay or is a timer instance, the execution of the consuming transition is time-triggered; otherwise, it is event-triggered. Task actions can be specified with attributes task id and task priority (line 5). The task id is a unique value of type **Tid** (line 6), a new type comparable to the SDL type **Pid**, and is only allowed in combination with **contTask**. By forking a task by

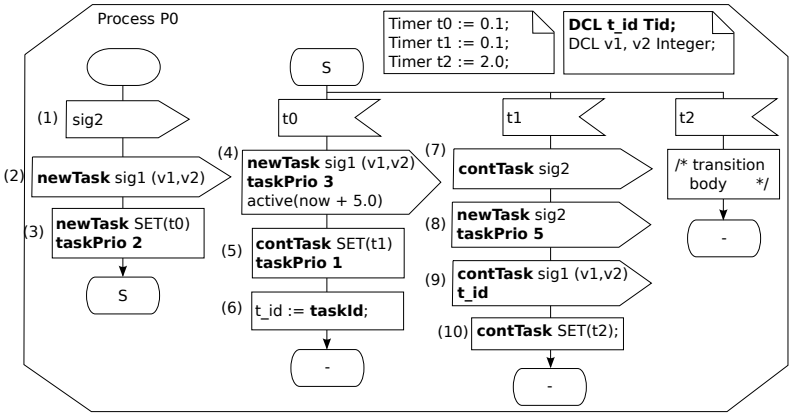

Fig. 2. Example of the usage of tasks in SDL

using **contTask** with a task id, the continuation of the task with the given task id is triggered. If **contTask** is used without task id, the task associated with the executed transition is continued. The specification of a task priority (line 7) is optional. Higher priority values mean lower task priority. If a new task is created without explicit specification of a task priority, the system assigns the predefined lowest priority (denoted as LOW_{predef} in Fig. 1). If **contTask** is used without priority, the task is continued with the same priority. Using **contTask** with priority sets the priority of the corresponding transition execution. Finally, **taskId** is a function returning the task id of the current task, which is `null` if the executed transition is not associated with a task. It can be used in *task assignments* to store the id of an existing task in a variable of type **Tid**.

```
1 <output body item> ::= [<task action>] <signal identifier>
      [<actual parameters>] [<task parameters>] [<activation
      delay>] [<signal priority>]
2 <set statement> ::= [<task action>] set <set body> [<task
     parameters>]
3
4 <task action> ::= newTask | contTask
5 <task parameters> ::= [<task id>] [<task priority>]
6 <task id> ::= < tid expression0>
7 <task priority> ::= taskPrio < Natural expression>
8
9 <imperative expression> ::= <now expression> | .... | <tid
     expression>
10 <tid expression> ::= taskId
```

Listing 1.1. Changes of the concrete SDL syntax (SDL signals and timers).

Figure 2 shows the application of the syntactical extensions in an SDL process P0. The SDL process is specified to generate the input queue of P1 in Fig. 1. Distributed over four transitions, there are four task creations, four task forkings,

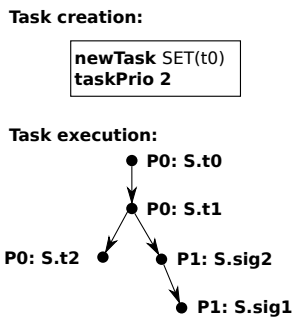

Fig. 3. Example of task creation and execution.

and one regular signal output, showing various ways to apply SDL tasks in a complex synthetic example: At (2), a task is created with the pre-defined lowest priority. The task created with an SDL timer at (3) is defined with priority 2 and is started with the transition consuming `t0` in S. This task is continued at (5) by setting another timer `t1`. At (4), a new task is created by a signal output with activation delay, thereby starting a time-triggered task in P1 remotely. The task assignment at (6) stores the identifier of the task dynamically associated with the execution of the current transition in variable `t_id` to be used in later transitions. The task executing the transition consuming `t1` is continued in (7) and (10) by using **contTask** without task id. At (9), the same task is continued by using the task identifier explicitly. As an example, Fig. 3 illustrates the resulting hierarchy of transition executions caused by the task creation at (3).

3 Implementation Aspects

Currently, we are in the process of completing the implementation of real-time tasks in our SDL tool chain. This section presents our implementation approach, and addresses implications of real-time tasks on transition scheduling.

3.1 Required Changes of the SDL Tool Chain and Limitations

Our SDL tool chain consists of the code generator *ConTraST* [11], the SDL Virtual Machine (SVM) implementation *SdlRE*, and the *SDL Environment Framework* (SEnF). It is compatible with the model-driven development approach [12] allowing automatic transformations of SDL specifications to platform-specific object files that can be deployed to various hardware platforms. We are currently supporting the Imote2 platform [13], Linux/PC, and various network simulators.

For the specification of SDL, we use the graphical editor of IBM's Rational SDL Suite [4]. To be compatible and to continue using its syntax and semantics analysis tool, we incorporate real-time tasks as annotations. These formal SDL comments are preserved when exporting to SDL/PR files.

By extending ConTraST, the annotations are considered during the transformation of SDL/PR to C++. In particular, actions for task creation and forking have to be attached to the corresponding signals when generating code of output and timer activation statements. Further extensions are required for SdlRE to generate unique task ids and to associate task attributes with SDL signals and transitions. Additionally, the selection of transitions has to be adapted according to the extended SDL semantics. We are furthermore going to change the scheduler of SdlRE to enforce task priorities system-wide (see Sect. 3.2).

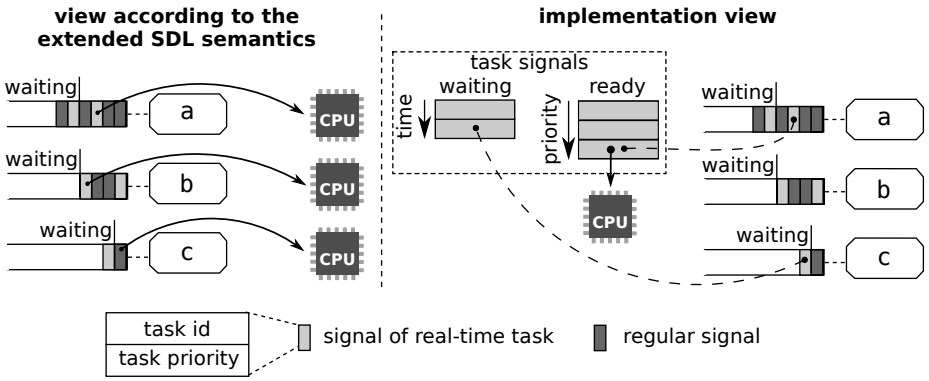

Fig. 4. Comparison of SDL's execution model and the implementation's model

In general, an implementation of real-time tasks has to face the challenge of an unbounded task id domain as introduced by the extended formal semantics. We overcome this problem by taking task ids from a *large* id pool and by reusing them from time to time. Though this is a limitation in theory, we expect that it has no practical relevance.

3.2 Scheduling of Real-Time Tasks

According to the semantics of SDL [14], all agents – SDL agents, SDL agent sets, and link agents – are executed concurrently. However, implementing SDL on real hardware requires serialization of agents. This serialization order is determined by a scheduler, which is in our tool chain part of SdlRE. The only scheduling constraint according to the SDL semantics is that every agent is eventually selected for execution. For real-time systems, this is not sufficient, as urgencies of transitions have to be taken into account.

With priorities of real-time tasks as introduced in Sect. 2, SDL has been extended to privilege important signals of a single agent. However, real-time tasks so far do not affect the global execution order of agents, because they are based on the same concurrent execution model of SDL. Dealing with task priorities system-wide is therefore left to the scheduler of the implementation.

In the left part of Fig. 4, execution according to the extended SDL semantics is illustrated. It is compared to the execution model of our implementation under development. According to the SDL semantics, every agent has its own input queue – separated into available signals and waiting signals – and processing unit. Though all agents consider task signals first and privilege available signals with highest task priority, the selection and execution of transitions of different agents is still independent of each other. In contrast, the implementation view on the right-hand side includes global queues of task signals to privilege the available signal with the highest task priority for execution on a single CPU. There are two queues of task signals: The *waiting queue* contains all signals

with an arrival time larger than `now`, e.g., non-expired timers, and is sorted by the availability time. All available task signals are in the *ready queue*, which is sorted by priority. The scheduler first searches for consumable task signals in the ready queue and selects the signal with the highest priority. If there is no such signal, an arbitrary agent is chosen to process non-task signals according to the standard SDL semantics.

Currently, the SdlRE scheduler only supports non-preemptive strategies. However, this is a design decision of our implementation and no general implication of SDL real-time tasks. Nevertheless, missing preemption may result in large queueing delays of signals with high task priority in the presence of long-running transitions. A way to deal with this problem is the temporary suspension of scheduling entities with low priorities. In [7], we have applied this idea to low-priority agents, thereby decreasing reaction times of single transitions significantly. By borrowing this approach to the scheduling of real-time tasks and by suspending real-time tasks based on task ids and priorities, reaction times of urgent real-time tasks that may consist of several transitions can be reduced.

4 Use of Real-Time Tasks in MacZ

This section illustrates the application of real-time tasks in the MAC layer protocol MacZ [15]. MacZ is a quality-of-service MAC protocol for wireless sensor networks providing tick and time synchronization, medium slotting, contention- and reservation-based medium access, and duty cycling.

Figure 5 presents a simplified excerpt of the architecture of MacZ's service layer. Processes in block `ContTxRx` are responsible for the contention-based transmission of data frames, i.e. they perform Carrier Sensing Multiple Access with Collision Avoidance (CSMA/CA; process `csma`) and maintain a Network Allocation Vector (process `nav`). In the block `ResTxRx`, reservation-based transmissions are processed. Depending on the synchronization (signal `Tick`), process `ctrl` in block `Controller` activates the transmission components in pre-configured slot regions by sending `Enable` signals. In addition to the service layer, Fig. 5 contains a single service user in block `ServiceUser` that is connected to the reservation-based transmission component.

To demonstrate how real-time tasks are used to improve the real-time behavior of MacZ, we specify two tasks. Task 1 is the activation of the contention-based component in the corresponding slot region. Task 2 shows the reservation-based transmission of sensor values.

Figure 6 shows the transitions executed during Task 1. The task consists of four transition executions and spans the processes `ctrl`, `contTxRx`, and `csma`. Its creation is triggered in `ctrl` when a synchronization tick is consumed. In this transition, there are two signal outputs, both starting a new task with priority 0 by sending `Enable` signals to `resTxRx` and `contTxRx`, respectively[2]. Both signals have two parameters stating the start and end time of the corresponding slot

[2] For simplicity, we assume that there is only one contention- and one reservation-based slot region. However, this is no general limitation of MacZ.

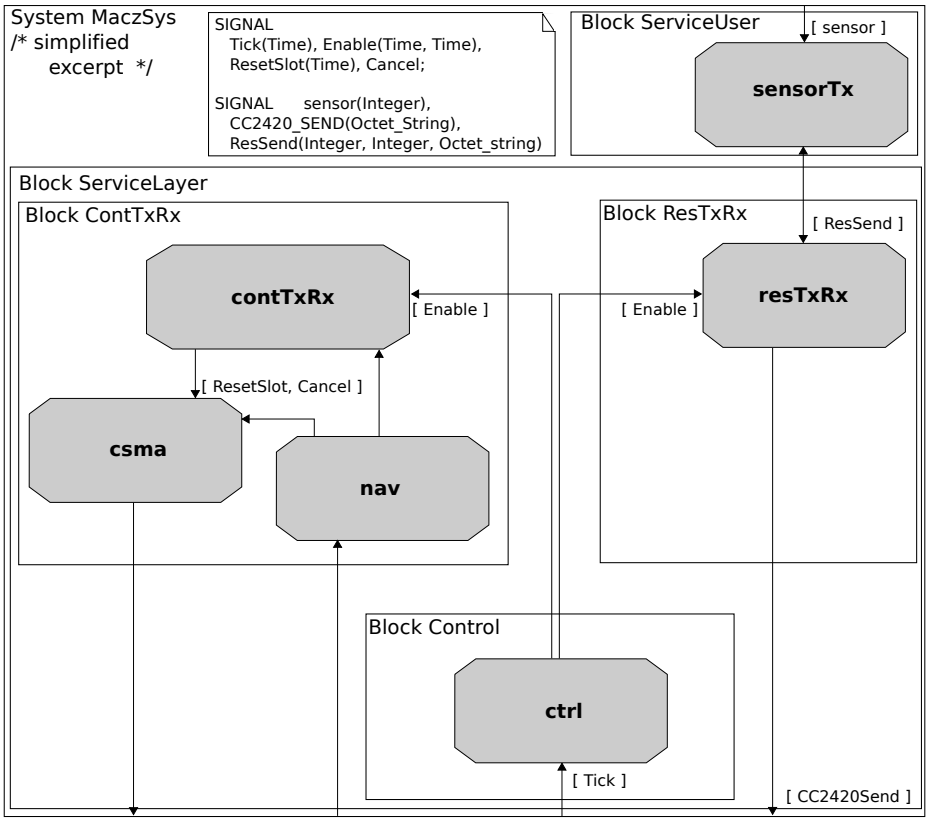

Fig. 5. Simplified excerpt of the SDL specification of MacZ with an example service user

region. In addition, the activation delay as introduced in SDL-2010 [10] is used to delay the signals' consumption to the start time of the slot region. The tasks are specified with highest priority, because slot region borders must be met accurately.

The relevant signal of Task 1 is the `Enable` signal to `contTxRx`. When the signal is consumed, task execution is started. In the transition consuming the signal, the associated real-time task is continued by sending a `ResetSlot` signal to the process `csma`, and by setting the `Disable` timer to the end of the slot region. Because no new task priority is given, the task priority remains 0. When `csma` receives `ResetSlot`, the start time of the slot region, which is required for slotted CSMA/CA, is set.

The real-time task is continued after expiration of the `Disable` timer in `contTxRx`. In the transition consuming the timer signal, it is checked whether there is a pending send job. If this is the case, the task is continued by sending a `Cancel` signal to `csma`, thereby stopping the transmission attempt in `csma` (not

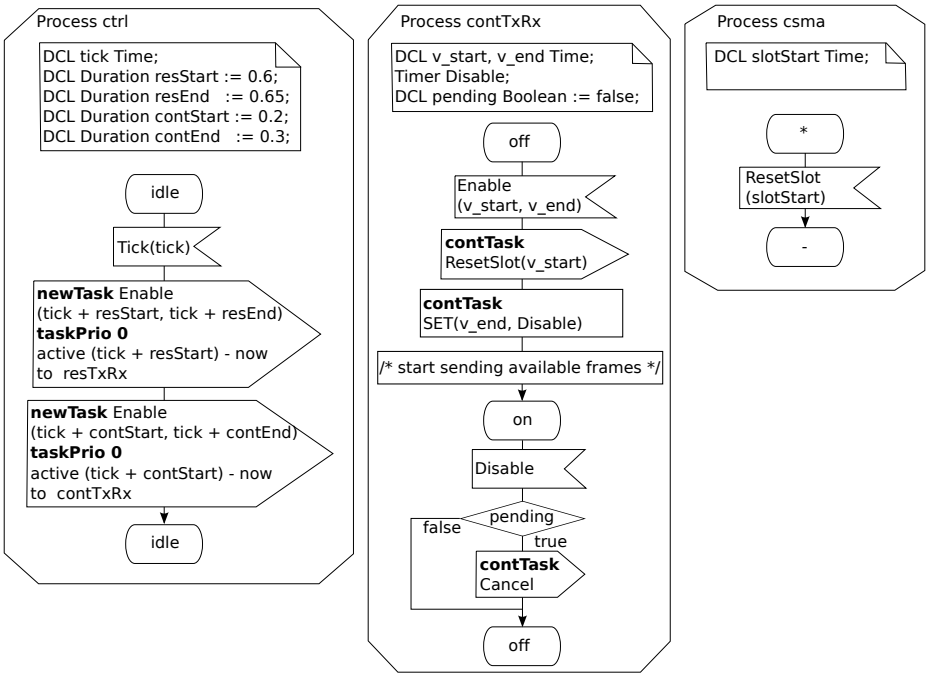

Fig. 6. Task 1: Activation of contention-based slot region

shown in figure). The task terminates as soon as its transition executions are finished, and there are no signals associated with the task.

Task 2 is illustrated in Fig. 7 and involves processes `sensorTx` and `resTxRx`. In process `sensorTx`, the task is created periodically by setting the timer `SendT` with task priority 3. This priority is sufficient, because we assume that the transmission of sensor values is not time-critical in the scenario, and that they are transmitted in the next reservation cycle if they do not arrive at the reservation-based transmission component in time. When consuming the `SendT` signal, the task is started and continued by sending a `ResSend` signal containing the destination's node id, a slot number[3], and the sensor data.

In the example, we assume that process `resTxRx` is not active (state `off`) when receiving `ResSend`, i.e. we are currently not within a reservation-based slot region. Thus, the MAC frame is prepared for transmission and placed in a local queue to be transmitted in the reserved slot. In addition, we keep the id of the task executing the transition by using the `taskId` literal. The task is continued when the reserved transmission slot is reached, which is indicated by the expiry of the `SendNext` timer. In the example, we assume that `SendNext` has been set by another task. In the transition, a `CC2420_SEND` signal continues the task by using the previous task id. In addition, the priority of the task is changed to the highest priority 0 to ensure that the frame transmission hits its slot boundary.

[3] In the example, we ignore that this is usually done using a reservation protocol.

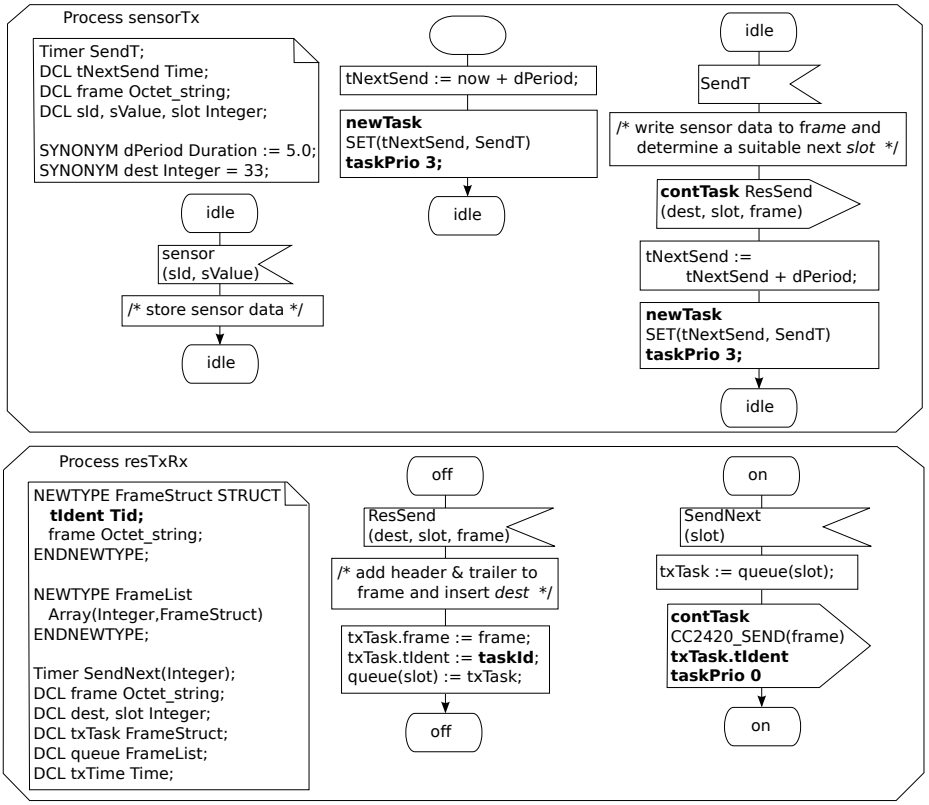

Fig. 7. Task 2: Reservation-based transmission of sensor values

5 Related Work

To the best of our knowledge, the concept of SDL real-time tasks as introduced in Sect. 2 has not been considered in the literature before. However, real-time tasks contain two aspects with existing related work: First, they influence the execution order of SDL transitions. We survey this aspect by looking at the activity thread model and at transition scheduling in SDL systems. Second, real-time tasks identify process-spanning functionalities, which we outline afterwards.

Activity Thread Model. An efficient way to implement node-internal signal transfer is the mapping onto method calls [16,17,18,19]. This approach is different from communication in SDL, since it is synchronous and blocking, and mixes communication and scheduling/execution of transitions [19]. However, in some circumstances, it is a simple, efficient, and standard-compliant way of implementing SDL.

In [16,17], the mapping of SDL onto the activity thread model is discussed. In an activity thread implementation, every input signal is realized by a corresponding procedure, i.e. a series of transitions leads to nested procedure calls. They are also common in manual protocol implementations for up- and downward communication in protocol stacks [20]. Similar to real-time tasks, activity threads state a special paradigm of event-driven implementation, in which not SDL processes but signals are treated as active entities.

Though an activity thread implementation is very efficient, it has several drawbacks. A main shortcoming is their limited applicability to systems with cyclic signal flows [18], which is partially solved in [16,17] by reordering output statements within transitions at compile time. However, several situations remain in which an activity thread implementation would lead to deadlocks or violations of SDL's semantics[4].

Compared to real-time tasks, activity threads do not add language expressiveness to SDL. Their improvements are limited to performance aspects without being capable of preferring urgent SDL transitions at run-time.

Transition Scheduling. According to SDL's semantics, all agents run asynchronously and concurrently that is not realizable on real hardware systems. Here, a scheduler must provide an adequate serialization of system initialization and execution, considering urgencies and priorities where specified.

In [8], Alvarez et al. present a preemptive execution model for SDL. Some details on their implementation are given in [21]. The execution model is based on dynamic process priorities that are derived from fixed transition priorities. One of the authors' objective is a real-time analysis of the system in order to check if the system meets its deadlines. To overcome schedulability problems that may be detected during this process, redesigning heuristics are presented.

The Cmicro integration, which is part of IBM Rational SDL Suite, supports the assignment of signal priorities [4]. By using a global signal queue and sorting signals according to their priority, the Cmicro scheduler selects the transition consuming the signal with highest priority. Thereby, different from SDL-2010 [1], signal priorities in Cmicro take precedence over availability time.

Compared to scheduling based on real-time tasks, process-based scheduling is very limited, because scheduling decisions are based on structural elements and not on functionally related transitions. Though signal priorities seem to be similar to real-time task priorities, they have two disadvantages: First, signal priorities are not sufficient to identify tasks, and therefore are less expressive than real-time tasks. Second, from a scheduling point of view, priorities are not passed on to output signals, i.e. there is no inheritance of priorities, thereby limiting their applicability if transitions are shared by several tasks.

[4] To overcome this problem, the authors suggest hybrid implementations, which use the activity thread model as well as the server model, which is a straight-forward implementation of the SDL semantics. By providing a control and specification language called iSDL, the implementor can choose between both models [17].

Design and Analysis Aspects. In [22], Kolloch et al. present a mapping of SDL systems to Real-Time Analysis Models (RTAMs) consisting of several independent analysis task precedence systems, each being triggered by an event. Based on the model, schedulability analysis with the earliest deadline first strategy are performed to the system. The authors' objective is not the improvement of SDL's expressiveness and, hence, they do not introduce the notion of task in SDL. However, the meaning of tasks in an RTAM is similar – yet less generic – to the concept of real-time tasks.

During the requirement phase, identification of system functionalities is often done by means of Message Sequence Charts (MSCs) [23]. Since a real-time task performs a specific system functionality, too, MSCs can be used to visualize them. There is some related work dealing with the automatic transformation of MSCs to SDL. For exmaple, [24] proposes a transformation for early performance predictions. Their approach is very limited – e.g., they do not support states – and the resulting SDL specification is not intended for further reuse. In [25], Khendek and Vincent address the enrichment of an existing SDL specification with new behavior defined by an MSC, e.g., by adding signals and transitions to the system. For this, they present a tool called MSC2SDL, which applies the transformations while preserving the existing behavior. In [26], an algorithm is presented building a complete SDL specification based on (High-level) MSCs and the architecture of the target design.

Though the objective of transformation approaches is completely different (they either want to enable analysis or achieve consistency of MSC and SDL specifications), there is also a similarity with real-time tasks, because in both cases, the SDL system is seen as composition of tasks. In general, such approaches have the disadvantage that they require knowledge of another language and special tool support. Because they are not intended for system implementations, their influence on the run-time behavior is very limited.

6 Conclusions

In this work, we have presented an extension of SDL to formally specify real-time tasks, a concept known from real-time systems. We have defined a real-time task in SDL to be a hierarchical order of executions of SDL transitions, which may span different SDL processes. We have defined syntactical extensions and their semantics, have outlined our implementation approach, and have demonstrated the use of real-time tasks in a complex MAC protocol.

Currently, we are in the process of completing the implementation of real-time tasks in our SDL tool chain. As soon as this is finished, we will run experiments in order to assess the benefits of the extension in terms of shorter and more predictable execution times.

So far, our notion of real-time task is restricted to SDL processes of a single SDL system. For a distributed implementation, an SDL system would typically be split into several interacting SDL systems, which would then be implemented on different nodes and executed under the control of local SVMs. This means

that a task may be executed on several nodes, and therefore has to be identified globally. We leave this aspect for our future work.

In our opinion, adding real-time tasks is a significant step towards making SDL a better design language for real-time systems and we are persuaded of SDL tasks being a candidate for inclusion in future SDL standards. Yet, hard real-time systems have further requirements that can still not be met. For instance, the problem of WCETs is an open one, and we feel that it can not be fully addressed in SDL. One reason is that it is not sufficient to consider WCETs of SDL transitions. In addition, the overhead created by running an implementation of the SVM must be considered. This overhead is, for instance, produced by selecting SDL transitions, and difficult to predict. Furthermore, the WCET of a medium priority task can not be predicted without making assumptions on the frequency and WCETs of high priority tasks. Therefore, we believe that another approach is needed, where, for instance, WCETs are measured at runtime to validate real-time requirements. Also, probabilistic WCETs may be an option.

Acknowledgments. This work is supported by the Carl Zeiss Foundation.

References

1. International Telecommunication Union (ITU): Z.100 – Specification and Description Language - Overview of SDL-2010 (2012), http://www.itu.int/rec/T-REC-Z.100-201112-I
2. SDL-RT Consortium: SDL-RT – Specification & Description Language – Real Time V2.2, http://www.sdl-rt.org/standard/V2.2/pdf/SDL-RT.pdf
3. PragmaDev SARL: Real Time Developer Studio, http://www.pragmadev.com/
4. IBM Corp.: Rational SDL Suite, http://www-01.ibm.com/software/awdtools/sdlsuite/
5. Kopetz, H.: Real-Time Systems – Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers (1997)
6. Krämer, M., Braun, T., Christmann, D., Gotzhein, R.: Real-Time Signaling in SDL. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 186–201. Springer, Heidelberg (2011)
7. Christmann, D., Becker, P., Gotzhein, R.: Priority Scheduling in SDL. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 202–217. Springer, Heidelberg (2011)
8. Álvarez, J.M., Díaz, M., Llopis, L., Pimentel, E., Troya, J.M.: Integrating Schedulability Analysis and Design Techniques in SDL. Real-Time Systems 24(3), 267–302 (2003)
9. International Telecommunication Union (ITU): Z.100 – Specification and Description Language, SDL (2007), http://www.itu.int/rec/T-REC-Z.100-200711-S
10. International Telecommunication Union (ITU): Z.101 – Specification and Description Language, Basic SDL-2010 (2012), http://www.itu.int/rec/T-REC-Z.101-201112-I
11. Fliege, I., Grammes, R., Weber, C.: ConTraST - A Configurable SDL Transpiler and Runtime Environment. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 216–228. Springer, Heidelberg (2006)
12. Gotzhein, R.: Model-driven by SDL – Improving the Quality of Networked Systems Development (Invited Paper). In: Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), pp. 31–46 (2007), http://vs.cs.uni-kl.de/en/publications/2007/Go07/Go07.pdf

13. MEMSIC Inc.: Data sheet – Imote2 Multimedia, http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html
14. International Telecommunication Union (ITU): Z.100 Annex F – SDL formal definition (2000), http://www.itu.int/rec/T-REC-Z.100-200011-S!AnnF1, http://www.itu.int/rec/T-REC-Z.100-200011-S!AnnF2, http://www.itu.int/rec/T-REC-Z.100-200011-S!AnnF3
15. Becker, P., Gotzhein, R., Kuhn, T.: MacZ – A Quality-of-Service MAC Layer for Ad-hoc Networks. In: Proceedings of the 7th International Conference on Hybrid Intelligent Systems (HIS 2007), pp. 277–282. IEEE Computer Society (2007)
16. Langendörfer, P., König, H.: Automated Protocol Implementations Based on Activity Threads. In: Proceedings of the Seventh Annual International Conference on Network Protocols (ICNP 1999), pp. 3–10. IEEE Computer Society (1999)
17. König, H., Langendörfer, P., Krumm, H.: Improving the Efficiency of Automated Protocol Implementations Using a Configurable FDT Compiler. Computer Communications 23(12), 1179–1195 (2000)
18. Sanders, R.: Implementing from SDL. Telektronikk 4 (2000), Languages for Telecommunication Applications. Telenor (2000), http://www.telektronikk.com/volumes/pdf/4.2000/Telek4_2000_Page_120-129.pdf
19. Bræk, R., Haugen, Ø.: Engineering Real Time Systems. Prentice Hall (1993)
20. Mitschele-Thiel, A.: Engineering with SDL – Developing Performance-Critical Communication Systems. John Wiley & Sons (2000)
21. Álvarez, J.M., Díaz, M., Llopis, L., Pimentel, E., Troya, J.M.: Deriving Hard Real-time Embedded Systems Implementations Directly from SDL Specifications. In: Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES 2001), pp. 128–133. ACM Press (2001)
22. Kolloch, T., Färber, G.: Mapping an Embedded Hard Real-Time Systems SDL Specification to an Analyzable Task Network - A Case Study. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 156–165. Springer, Heidelberg (1998)
23. International Telecommunication Union (ITU): Z.120 – Message Sequence Chart (MSC) (February 2011), http://www.itu.int/rec/T-REC-Z.120-201102-I/en
24. Dulz, W., Gruhl, S., Lambert, L., Söllner, M.: Early Performance Prediction of SDL/MSC Specified Systems by Automated Synthetic Code Generation. In: SDL 1999: The Next Millennium, pp. 457–472. Elsevier Science (1999)
25. Khendek, F., Vincent, D.: Enriching SDL Specifications with MSCs (2000), http://www.irisa.fr/manifestations/2000/sam2000/PAPERS/P16-Khendek2.ps.gz
26. Khendek, F., Zhang, X.-J.: From MSC to SDL: Overview and an Application to the Autonomous Shuttle Transport System. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 228–254. Springer, Heidelberg (2005)

Appendix A. Semantical Extensions of SDL

To formally incorporate the execution of real-time tasks in SDL, we have modified the dynamic SDL semantics in SDL-2000, Z.100 F3 [14], which is the latest approved version of SDL's formal ASM semantics.

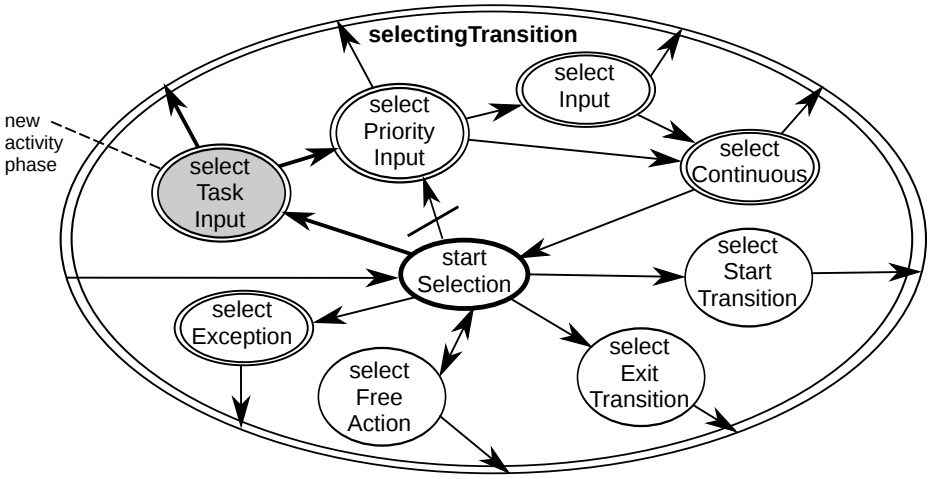

Fig. 8. Extended activity phases of SDL agents when selecting the next transition to be executed [14]: Before searching for transitions of priority inputs, an agent first searches for transitions of task inputs.

Lines 2-5 of List. 1.2 define new ASM domains TId, TaskPriority, and TaskAction. This is followed by new ASM functions to determine task ids of signal instances and SDL agents, and priorities of tasks. The task id of an agent is initialized with **null** during the initialization of the agent's control block. During a transition, it is set to the task id of the consumed signal. Since this modification is minor, it is not shown in the listing.

Further modifications shown in lines 13-32 apply to output and set actions, which are extended by task action, task id, and task priority. These values are used in the new ASM macro ConfigTask (lines 35-50), which sets task id and task priority of signals created in ASM macros SignalOutput and EvalTimer.

An important modification concerns the selection of transitions. Our approach is to give preference to transitions triggered by a signal that is associated with a task. This means that we precede the transition selection phase of Z.100 (sketched in lines 53-68), which considers priority inputs, regular inputs, continuous signals, and spontaneous signals, by *task inputs*. For a given SDL agent, we search the entire input queue of arrived signals in order to determine the first task input with highest task priority, i.e. the active signal with the lowest task priority value. If there is a task input, the selection phase terminates, and the corresponding transition is selected for execution. Otherwise, the selection phase is continued with the priority input selection as described in Z.100 (see also Fig. 8). Thus, transitions associated with real-time tasks always have preference over regular transitions. Also, the extension is compatible with Z.100, as the semantics of SDL systems without real-time tasks remains the same.

```
1  // New domains
2  shared domain TID
3              initially TID = { null }
4  TASKPRIORITY =def NAT ∪ { lowestPriority }
5  TASKACTION =def { newTask, contTask }
6
7  // New functions
8  shared tId : SIGNALINST → TID
9  controlled tId : SDLAGENT → TID
10 controlled taskPriority : TID → TASKPRIORITY
11
12 // Changed tuples
13 OUTPUT =def SIGNAL ×VALUELABEL* ×VALUELABEL ×VIAARG ×
       TASKACTION ×TID ×TASKPRIORITY ×CONTINUELABEL
14 SET =def TIMELABEL ×TIMER ×VALUELABEL* ×TASKACTION ×TID ×
       TASKPRIORITY ×CONTINUELABEL
15
16 // Changed macros regarding ordinary signals
17 SIGNALOUTPUT(s:SIGNAL, vSeq:VALUE*, toArg:TOARG, viaArg:VIAARG,
       taskAction:TASKACTION, taskId:TID, taskPriority:TASKPRIORITY) ≡
18     ...
19         choose g: g ∈ Self.outgates ∧ Applicable(s, TOARG, VIAARG, g,
               undefined)
20             extend PlainSignalInst with si
21                 ...
22                 CONFIGTASK(si, taskAction, taskId, taskPriority)
23                 INSERT(si, now, g)
24             endextend
25         endchoose
26
27 // Changed macros regarding timers
28 SETTIMER(tm:TIMER, vSeq :VALUE*, t:TIMER, taskAction:TASKACTION, taskId:
       TID, taskPriority:TASKPRIORITY) ≡
29     let tmi = mk–TimerInst(Self.self, tm, vSeq ) in
30         ...
31         CONFIGTASK(tmi, taskAction, taskId, taskPriority)
32     endlet
33
34 // New help macro
35 CONFIGTASK(si:SIGNALINST, taskAction:TASKACTION, taskId:TID, taskPriority:
       TASKPRIORITY) ≡
36     if taskAction = newTask then
37         extend TID with tId
38             si.tId :=tId
39             si.tId.taskPriority :=taskPriority
40         endextend
41     elseif taskAction = contTask then
42         if taskId = null then
43             si.tId :=Self.tId
44         else
```

```
45              si.tId :=taskId
46          endif
47          if taskPriority ≠ lowestPriority then
48              si.tId.taskPriority :=taskPriority
49          endif
50      endif
51
52  // Sketch of changed macros regarding transition selection
53  AgentMode =def { ..., selectTaskInput, ... } // New element added
54
55  SelectTransitionStartPhase ≡
56      if Self.currentExceptionInst ≠ undefined then
57      ...
58      else
59          Self.inputPortChecked :=Self.inport.queue
60          Self.agentMode3 :=~~selectPriorityInput~~ selectTaskInput
61          Self.agentMode4 :=startPhase
62      endif
63
64  SelectTransition ≡
65      ...
66      elseif Self.agentMode3 = selectTaskInput then
67          SelectTaskInput
68      ...
```

Listing 1.2. Changes to the formal semantics of SDL-2000.